

ISSN 2745-2336
9 772745 233609
celer
Revista
CelerSMS
Artículos Técnicos
Año 1 / N° 2 / 2021/12/31
BOGOTÁ
Seguridad en los Teléfonos Android

CONTENIDO

La cadena es tan fuerte como el eslabón más débil



«La seguridad no es un producto, es un proceso»

Bruce Schneier, 2011[A]

Los problemas de seguridad en teléfonos móviles no eran tan grandes hace 10 años, ya que el uso de dichos teléfonos para almacenar e intercambiar información sensible no era común en ese entonces. La frase célebre de Bruce Schneier hace referencia a las redes informáticas, pero se aplica perfectamente a los temas que abordaremos en esta edición.

DIRECTORIO

La revista CelerSMS es una publicación temática semestral acerca de las tecnologías de telecomunicaciones, plataformas móviles y programación de sistemas. Todo el material publicado en CelerSMS está disponible de manera gratuita y no incluye información comercial o publicitaria. La revista puede ser descargada de la dirección web oficial:

www.celersms.com/revista/

Estos contenidos también pueden ser consultados por medio de Google News:

news.google.com/publications/CAAqBwgKMMKklQswu4WrAw



Las opiniones expresadas por los autores no necesariamente reflejan la postura del editor de la publicación.

Esta revista puede ser reproducida con fines no comerciales, siempre y cuando se cite la fuente completa y su dirección electrónica.

Información de contacto:

info@celersms.com
(+57)3023436167
BOGOTÁ

ISSN 2745-2336
ISNI 0000 0005 0265 593X



EDITORIAL

Sabía qué hace 10 años los teléfonos celulares ya eran usados para realizar transacciones bancarias? Los servicios de banca móvil ya existían inclusive antes de la era de los teléfonos inteligentes. Dichos servicios utilizaban la tarjeta SIM para encriptar los mensajes. Hasta la fecha no se conocen casos de fraude con este servicio, al menos en Colombia, contrario a los servicios bancarios por medio de apps o páginas web. Entonces, es válido preguntarse por qué los servicios móviles de hoy, luego de una década de evolución, no son más seguros.

Según Asobancaria, los casos de fraude cibernético van en aumento. Cada año crecen la cuantía de las pérdidas y la frecuencia con la que se presentan estos hechos.[1]

El robo de información personal y la invasión de la privacidad también son cada vez más frecuentes. De hecho, muchas veces los usuarios mismos publican información sensible. Por ejemplo, compartir la fecha de cumpleaños o el lugar de nacimiento incrementa la vulnerabilidad. En los años 2000 uno de los bancos más importantes de la región permitía realizar transferencias a cuentas en el exterior indicando únicamente el número de documento de identidad del titular. Hoy en día se aplican protocolos más seguros, pero aún existen muchos esquemas de fraude que se basan en la información que los usuarios desprevenidos publican en redes sociales. También nos podemos preguntar si en estos casos el manejo de la información personal es responsabilidad del usuario. El desarrollador del servicio puede y debe prever que los usuarios no siempre utilizan las mejores prácticas en el manejo de su información personal.

Muchas publicaciones ya exponen esta problemática desde el punto de vista del usuario. Generalmente se recomienda evitar descargar aplicativos de fuentes no confiables, visitar sitios desconocidos en internet, dejar la sesión abierta en los aplicativos que manejan información sensible, etc. En esta edición pretendemos abordar la misma problemática, específicamente en Android, pero desde el punto de vista del desarrollador de aplicaciones.

Frecuentemente las medidas que mejoran la seguridad afectan la comodidad del usuario, por ejemplo: cambiar la contraseña periódicamente, usar contraseñas con un alto grado de complejidad, digitar claves dinámicas, etc. Los usuarios no suelen reconocer la prevención de los intentos de *hackeo* de sus datos. En cambio, la necesidad de pulsar un botón de más o memorizar una clave siempre generan molestias.

¿Qué tan necesaria es la información sensible que obtienen y transmiten las aplicaciones? A veces, las apps obtienen el número de serie o el número de teléfono simplemente para identificar al usuario de manera única. Existen alternativas que no implican obtención de información sensible, como se expuso en la edición anterior.[2] Reducir al mínimo posible el manejo de información sensible mejora la seguridad de la aplicación.

El *boom* de la transformación digital ha incrementado el uso de servicios móviles, incluyendo los que ofrecen operaciones que hasta hace un par de años solamente podían realizarse de manera presencial con documentos en físico. Por ejemplo, ya existen servicios para adquirir créditos bancarios o comprar seguros desde el teléfono móvil, sin salir de casa. Además, en algunos países ya existen sistemas de

---

[1] *Alerta por incremento del fraude bancario por los canales digitales* (2021-07-01) https://www.eltiempo.com/economia/sector-financiero/fraude-bancario-modalidades-usadas-para-robar-dinero-600190

[2] Celer, Victor (2021-05-30) *Generación de Identificadores Únicos de Dispositivo Android* https://www.celersms.com/device-id-android-es.htm

registro electoral y votación electrónica desde el teléfono, inclusive para elecciones presidenciales.[3] Estos servicios hacen uso de tecnologías de verificación de identidad que permiten confirmar la validez de los documentos físicos, asociados con una persona real. Esta tecnología se conoce comercialmente como KYC (en inglés, *Know Your Customer*).[4] Seguramente estos servicios móviles se volverán cada vez más populares en el mundo. La seguridad es el mayor reto para vencer las barreras en la adopción masiva de estos servicios innovadores.

Los sistemas de pago con código QR o con tecnología NFC (en inglés, *Near Field Communication*)[5] tuvieron que superar retos similares.

Es interesante notar que todas estas tecnologías fueron introducidas inicialmente de manera selectiva. Por ejemplo, NFC se usó para pagos de monto mínimo en ambientes controlados. A medida que se lograba mitigar el riesgo de fraude se ampliaba el alcance de estos servicios. Por lo tanto, mejorar la seguridad para una tecnología existente puede llegar a ser más rentable que introducir una tecnología nueva.

Conforme se extiende el alcance de un sistema seguro, el sistema también se vuelve más atractivo para los estafadores. Por lo tanto, la seguridad de estos sistemas debe permanecer bajo revisión constante. Es importante implementar mecanismos de mitigación, los cuales se puedan activar oportunamente en caso de una emergencia de seguridad. Por ejemplo, si se compromete la seguridad de un certificado digital, debe ser posible revocar dicho certificado. Muchas de las crisis de seguridad conocidas tuvieron gran impacto porque carecían de mecanismos de mitigación.

La reputación de un producto puede quedar destruida a causa de una falla de seguridad. Por ejemplo, si buscamos en Google cualquier aplicación móvil popular y descubrimos una publicación acerca de algún problema de seguridad, seguramente tendremos preferencia por otra aplicación equivalente, la cual no haya tenido fallas de seguridad conocidas.

La revista CelerSMS desea a sus lectores un año 2022 seguro, lleno de éxitos tecnológicos y aplicaciones de excelente calidad.

Vladimir Kameñar
Editor

[3] Palmer, Mark (2020-10-29) *Is voting by mobile phone the future of democracy?* (en inglés) https://www.thearticle.com/is-voting-by-mobile-phone-the-future-of-democracy

[4] https://es.wikipedia.org/wiki/Conozca_a_su_cliente

[5] https://es.wikipedia.org/wiki/Near_Field_Communication



Seguridad vs. Usabilidad en las Apps Android

Miguel Angel Cano

Uno de los dilemas en el desarrollo de apps es el balance entre la facilidad de uso y la carga funcional. Esta última representa el conjunto de los casos de uso que ofrece la aplicación. Entre más grande el conjunto, más compleja la interacción con el usuario.

Muchas de las funciones de seguridad, como la autenticación del usuario o la confirmación de una operación sensible, implican acciones adicionales que el usuario debe realizar. Estas acciones pueden ser la lectura de la huella dactilar, el ingreso de una clave, etc. Estas acciones también son parte de la carga funcional y afectan la usabilidad[6] de la aplicación.

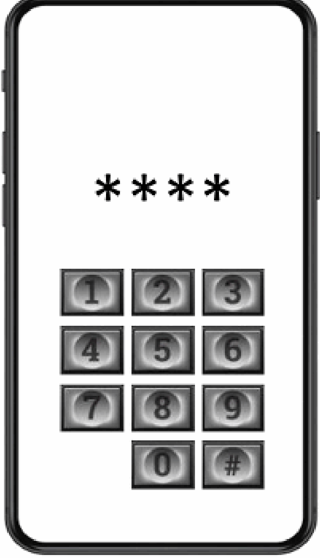
centenares de opciones, muchas de las cuales requieren conocimientos técnicos. Lograr que la experiencia de usuario (UX) sea intuitiva en una aplicación compleja es un reto importante, inclusive para los desarrolladores de Google.

Se dice que la interfaz de usuario es intuitiva si el usuario promedio puede comprenderla fácilmente, sin capacitación previa. No se debe confundir los conceptos de interfaz intuitiva e interfaz fácil de usar, aunque muchas veces sean lo mismo. Por ejemplo, ingresar una clave por medio de unas perillas giratorias que simulan la cerradura de una caja fuerte puede ser intuitivo, pero más complejo que usar el teclado numérico para ingresar un PIN.

¿Qué es una interfaz de usuario intuitiva?

Los ejemplos de aplicaciones más simples incluyen la linterna y la brújula. La interacción con el usuario es simple porque la funcionalidad de la aplicación es mínima. Probablemente nadie ha tenido que leer un manual para aprender a manipular la linterna o jugar *Tetris*.[7] En cambio, la aplicación de configuración de Android tiene







[6] *Definición de usabilidad por FundéuRAE* https://www.fundeu.es/consulta/usabilidad-2438/

[7] *Tetris, videojuego de lógica creado por Aleksei Pazhitnov* https://es.wikipedia.org/wiki/Tetris

Una interfaz de discado con dial rotatorio o disco de marcar,[8] la cual se usó en todos los teléfonos analógicos, es muy simple, pero hoy en día podría no ser intuitiva, ya que muchas personas no han visto o no recuerdan los teléfonos antiguos.

Interfaces web-app

Actualmente es común el uso de interfaces de tipo *web-app* dentro de las aplicaciones móviles. También se utilizan interfaces híbridas, en las cuales una parte de la interfaz es de tipo web-app y otra parte es nativa. Las interfaces de tipo web-app son generadas por un servidor web remoto y se despliegan dentro de la aplicación de una manera similar a los navegadores web, como Chrome. La popularidad de este tipo de interfaces se debe principalmente a 2 factores, a saber:

- Es posible realizar cambios en la interfaz de manera centralizada, sin publicar una nueva versión de la app y esperar que los usuarios descarguen la actualización.
- Se puede usar la misma base de código para la interfaz web y la app. Esto reduce los costos de desarrollo y mantenimiento.

Desde el punto de vista de seguridad, las interfaces web-app también pueden tener ventajas:

- Si una posible vulnerabilidad puede ser corregida a nivel de web-app, sin modificar el código nativo, el tiempo de mitigación se reduce considerablemente. Por lo tanto, el impacto también se reduce.
- Es más difícil *hackear* una solución que se ejecuta en un servidor remoto, ya que sólo se tiene acceso a la interfaz de usuario de la misma.

Los servidores web no son invulnerables. Una web-app también puede ser insegura. Sin embargo, en general el código que se ejecuta remotamente en la nube puede estar más protegido que el código nativo que se ejecuta localmente.



La experiencia de usuario con las interfaces web-app suele ser menos óptima, ya que la descarga de la interfaz puede tomar un tiempo, especialmente si la conexión de Internet es lenta. El uso de Internet, además de generar un consumo de datos, produce un incremento en el uso de batería en el teléfono. Se recomienda considerar el uso de interfaces web-app solamente si la aplicación implica conectividad, como las aplicaciones bancarias, redes sociales, etc.

Seguridad vs. comodidad

El mayor reto está en mejorar la seguridad de un servicio sin que estas mejoras generen una mala experiencia de usuario. Por ejemplo, el uso de la biometría, como la huella dactilar o reconocimiento facial, en lugar de memorizar claves, es uno de los avances tecnológicos que han mejorado la experiencia de usuario. El uso de biometría no implica desarrollo de complejos algoritmos de inteligencia artificial. Estos algoritmos ya están disponibles a nivel de hardware o sistema operativo del teléfono.[9]

Otro ejemplo son los sistemas de autenticación basados en patrones de comportamiento del usuario.[B] Estos sistemas estudian la manera como el usuario interactúa con el teléfono, lo cual incluye la manipulación de la pantalla táctil (posición, velocidad, ángulo de deslizamiento), frecuencia y duración de uso, entre otras variables. Ya existen implementaciones comerciales con este tipo de sistemas. De hecho, la idea de perfilar el comportamiento del usuario con fines de autenticación no es nueva. Las implementaciones experimentales han existido al menos desde 2016.[C]

[8] https://es.wikipedia.org/wiki/Disco_de_marcar
[9] *Autenticación biométrica en Android* https://developer.android.com/training/sign-in/biometric-auth

Estas implementaciones han demostrado tener una efectividad superior al 75%. Al combinar estos perfiles de comportamiento con otros factores, como biometría y claves tradicionales, se logra un mayor nivel de confiabilidad.

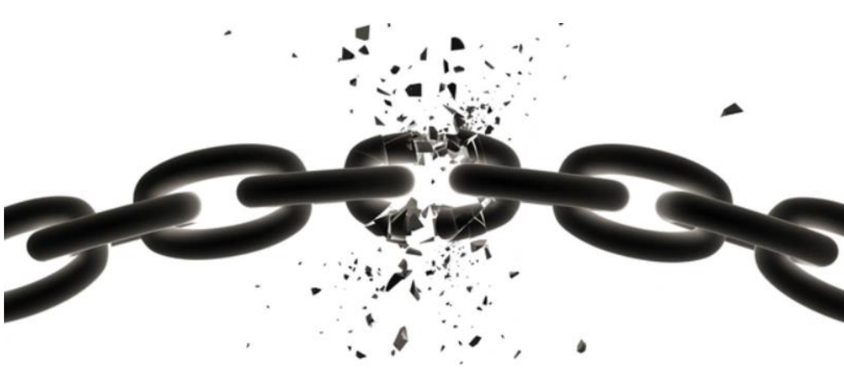
Sin embargo, es importante recordar que los usuarios pueden cambiar sus hábitos. Además, la autenticación es solamente uno de los aspectos que abarca la seguridad de un servicio. La suplantación de la identidad del usuario es solamente uno de los métodos que utilizan los estafadores para cometer fraude. Otros métodos consisten en interceptar los canales de comunicación explotando posibles vulnerabilidades, la ingeniería inversa, etc.

El eslabón más débil

Thomas Reid, filósofo escocés, escribió que «*una cadena es tan fuerte como su eslabón más débil*». Esta frase se aplica perfectamente en materia de seguridad informática, incluyendo los servicios móviles. Por lo tanto, es importante auditar constantemente cada uno de los módulos o procesos que intervienen en el servicio. Reducir al mínimo posible los eslabones de la cadena también ayuda a minimizar los riesgos.

El eslabón más débil puede ser el usuario, quien anota su contraseña y la guarda en un lugar visible para no olvidarla. El uso de protocolos poco seguros por compatibilidad con dispositivos antiguos o versiones previas del aplicativo también se puede convertir en una vulnerabilidad.

Un colega que desarrolla drivers para Windows dice que a veces el cambio más inofensivo puede ocasionar efectos desastrosos. El desarrollo de apps también es susceptible a este tipo de situaciones, aunque en menor medida. Un control estricto de cambios y versiones, combinado con pruebas de



control de calidad exhaustivas, reducen la probabilidad de errores, incluyendo los errores de seguridad.

En las grandes empresas de software existe el cargo de especialista en seguridad u oficial de seguridad de la información.[10] Contar con recursos dedicados a los tema de seguridad ayuda a reducir los riesgos. La seguridad es un aspecto menos visible que la usabilidad o la carga funcional, por lo que no siempre recibe la misma atención durante el ciclo de desarrollo. Además, los desarrolladores no siempre son expertos en seguridad. Por lo tanto, si el equipo de desarrollo no cuenta con especialistas en seguridad, se debe buscar capacitar a los desarrolladores. Muchas de las vulnerabilidades que se detectan en las apps se deben a desconocimiento o descuido.

---

10 https://es.wikipedia.org/wiki/Oficial_de_seguridad_de_la_informaci%C3%B3n



Control de Permisos Android

Victor Celer

El control de permisos en las primeras versiones de Android consistía únicamente en declarar dichos permisos dentro de *AndroidManifest.xml*. Hoy en día se mantiene la misma modalidad para los permisos menos ofensivos. En cambio, los permisos más sensibles, como el acceso a la ubicación, requieren aprobación expresa de parte del usuario. Algunos permisos, como el envío de SMS, requieren aprobación selectiva por parte del equipo de Google Play para la publicación en línea. Además, los aplicativos firmados por el operador móvil tienen mayores privilegios. Esto se conoce como privilegios de operador y se usa para configurar el dispositivo (por ejemplo, los servicios de VoLTE). Los permisos pueden funcionar de manera temporal o permanente. Además, algunos permisos pueden funcionar de manera diferente si el servicio se encuentra en segundo plano.

Permisos de tiempo de instalación

Los permisos de tiempo de instalación, como su nombre lo indica, se otorgan en el momento de la instalación de la aplicación. Un ejemplo de permiso de tiempo de instalación es el permiso para acceder a internet.

Para hacer uso de un permiso de tiempo de instalación simplemente se debe declarar dicho permiso en *AndroidManifest.xml*, por ejemplo:

```
<uses-permission
 android:name="android.permission.INTERNET" />
<uses-permission
 android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

Estos permisos son visibles para el usuario en la tienda en línea. Sin embargo, muchas veces el usuario no presta atención a estos permisos durante la instalación de la app. Por eso, los permisos que se obtienen de esta manera tienen un nivel de impacto bajo desde el punto de vista de seguridad.

Los permisos de tiempo de instalación se adjudican de manera permanente.

Permisos de tiempo de ejecución

En cambio, los permisos más sensibles requieren acciones adicionales a la declaración dentro del XML. Un ejemplo son los permisos de localización:

```
<uses-permission
 android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission
 android:name="android.permission.ACCESS_FINE_LOCATION" />
```

El permiso `ACCESS_COARSE_LOCATION` aplica para las apps que requieren acceso a la localización aproximada del dispositivo. La localización aproximada generalmente se obtiene por medio de los servicios de red. En cambio, el permiso `ACCESS_FINE_LOCATION` se usa para obtener la localización precisa, la cual generalmente hace referencia a GPS. Google recomienda incluir ambos

permisos para que el usuario pueda decidir si se aplica la localización aproximada o precisa, lo cual es posible a partir de Android 12.[11]

Sin embargo, solamente declarar estos permisos no es suficiente. A partir de Android 6 se debe solicitar estos permisos en tiempo de ejecución. De esta manera el usuario tiene mayor conciencia de los permisos sensibles y puede no autorizarlos.

Google recomienda utilizar las clases auxiliares `ContextCompat` y `ActivityCompat` para validar y solicitar los permisos en tiempo de ejecución. Por ejemplo:

```
import androidx.core.content.ContextCompat;
import androidx.core.app.ActivityCompat;

String COARSE  = "android.permission.ACCESS_COARSE_LOCATION";
String FINE = "android.permission.ACCESS_FINE_LOCATION";
int LOCRESULT = 123;

Context ctx = getApplicationContext();
if(ContextCompat.checkSelfPermission(ctx, FINE) != 0 &&
   ContextCompat.checkSelfPermission(ctx, COARSE) != 0){
      ActivityCompat.requestPermissions(this,
         new String[]{ FINE }, LOCRESULT);
}
```

Este código verifica si alguno de estos 2 permisos ya se encuentra aprobado. En caso contrario se despliega un cuadro solicitando que el usuario conceda los permisos. Nótese que la verificación de permisos se realiza sobre el contexto asociado con el proceso, en el cual se está ejecutando la aplicación. Esto es porque internamente Android asocia los permisos con el ID de proceso (PID) y el ID de usuario (UID). En cambio, la solicitud de permisos recibe un parámetro que representa la actividad (`Activity`) actual, ya que puede desplegar un elemento interactivo para que el usuario apruebe la solicitud. Cuando el usuario apruebe o desapruebe la solicitud de permisos, se invocará el método `onRequestPermissionsResult` sobre esta misma actividad:

```
@Override
public void onRequestPermissionsResult(
   int req, String perms[], int[] res){
   if(req == LOCRESULT
      && res.length > 0 && res[0] == 0){
      // permisos concedidos
   }
}
```

Es opcional procesar la notificación del resultado. La aprobación se realiza de manera asíncrona. Por lo tanto, procesar la notificación permite implementar una lógica condicional según la decisión del usuario. Por ejemplo, si el usuario decide no aprobar los permisos, la aplicación podría desplegar un mensaje de error o implementar una funcionalidad alterna.

Si no desea utilizar librerías auxiliares **AndroidX** (*Android Extension Library*),[12] es posible implementar la misma lógica por medio de las API estándar de Android. Esta opción no es recomendada por Google y podría dejar de funcionar en alguna versión futura de Android.

```
import android.os.Process;
import android.app.Activity;

String COARSE  = "android.permission.ACCESS_COARSE_LOCATION";
String FINE = "android.permission.ACCESS_FINE_LOCATION";
int LOCRESULT = 123;

int pid = Process.myPid(), uid = Process.myUid();
if(checkPermission(FINE, pid, uid) != 0 &&
   checkPermission(COARSE, pid, uid) != 0){
      requestPermissions(new String[]{ FINE }, LOCRESULT);
}
```

[11] *Permisos de ubicación*
https://developer.android.com/training/location/permissions

[12] *Descripción general de AndroidX*
https://developer.android.com/jetpack/androidx

En ambos casos es importante agregar la verificación de versión de SDK y un manejo mínimo de errores con un bloque try/catch, como sigue:

```
if(Build.VERSION.SDK_INT > 22){
   try{
     // verificar permisos de localización
   }catch(Exception exc){
      Log.e("App", "exception", exc);
   }
}
```

De esta manera, el código de validación de permisos se ejecuta solamente en Android 6 o posterior (SDK mayor que 22), ya que en versiones anteriores estos permisos eran otorgados en tiempo de instalación.

Recuerde que a partir de Android 12 se recomienda separar los permisos de localización aproximada y localización precisa para que el usuario pueda elegir uno de los 2 métodos, ambos o ninguno.

Para habilitar otros permisos de tiempo de ejecución se utiliza la misma metodología. Previo a la solicitud de los permisos, se recomienda desplegar un cuadro informativo explicando por qué se requieren dichos permisos. Esto ayuda a convencer al usuario para que no bloquee los permisos.

Los permisos de tiempo de ejecución pueden ser revocados automáticamente, luego de un período de inactividad. Esto sucede a partir de Android 11. Si la app deja de utilizarse durante un tiempo largo (varios meses), los permisos pueden ser revocados automáticamente.[13] Por lo tanto, la aplicación debe verificar los permisos cada vez que los va a utilizar.

Privilegios de operador

A partir de Android 5.1 se introdujo una funcionalidad conocida como privilegios de operador (en inglés, *Carrier Privileges*).[14] Esta funcionalidad permite otorgar permisos especiales para las aplicaciones cuya firma digital concuerda con la lista de firmas instaladas en la SIMcard. Se asume que solamente el operador móvil o el fabricante de la SIMcard pueden modificar la lista de firmas en la SIMcard. Un proveedor de aplicaciones independiente no tiene acceso a las llaves administrativas de la SIMcard. Por lo tanto, no puede manipular la lista de firmas.

Lo que se almacena en la SIMcard es un valor hash (SHA1) de la clave, no la clave misma. Opcionalmente, junto con este hash, se puede incluir el nombre del paquete de la aplicación, correspondiente al parámetro `package` dentro de *AndroidManifest.xml*. Esto puede ser útil si la misma firma se usa en múltiples aplicaciones, pero los privilegios de operador son requeridos solamente en algunas.

Existen 2 métodos para almacenar la lista de firmas en la SIMcard:

- Por medio de un aplicativo llamado ARA, el cual se ejecuta dentro de la SIMcard. Este aplicativo se encuentra definido en el estándar **GPD_SPE_013**.[D]
- Dentro de un archivo llamado ARF definido en el estándar **PKCS #15**.[E]

ARF es soportado a partir de Android 7 y se usa si no se encuentra ARA. Dado que ARA tiene prioridad y es soportado a partir de Android 5.1, este método es más usado que ARF.

Actualmente la gran mayoría de las SIMcard soportan los estándares *Global Platform*. Esto no garantiza que el aplicativo ARA se encuentre disponible en todas las SIMcard, ya que no todos los operadores hacen uso de esta funcionalidad. Una versión de ARA de código abierto se puede encontrar aquí:



El aplicativo **ARA-M** de Bertrand Martel es gratuito y compatible con la mayoría de las SIMcard.

[13] *Actualizaciones de permisos en Android 11* https://developer.android.com/about/versions/11/privacy/permissions

[14] https://source.android.com/devices/tech/config/uicc

Los operadores móviles acostumbran utilizar versiones de ARA suministradas por los fabricantes de SIMcard. Una de las razones es que los aplicativos genéricos pueden ocasionar uso intensivo de la memoria no volátil de la SIMcard, lo cual acorta su vida útil.



El proyecto **CoIMS** contiene una guía para instalar ARA-M. Además, incluye consejos para depurar errores comunes.

Para instalar un aplicativo en la SIMcard se debe contar con las llaves administrativas. El dilema está en que las SIMcard de pruebas, las cuales se comercializan para propósitos de ingeniería e incluyen llaves administrativas, generalmente no incluyen suscripción móvil. En cambio, las SIMcard que ofrecen los operadores móviles no incluyen llaves administrativas. Por lo tanto, las posibilidades de utilizar esta tecnología con una suscripción móvil real son limitadas. Generalmente se requiere una cooperación estrecha con el operador móvil para lograr aprovisionar las firmas del desarrollador en las SIMcard de los usuarios móviles.

A continuación se presenta la lista con algunas funcionalidades disponibles bajo privilegios de operador:

- Obtener el ID de la suscripción móvil (IMSI).
- Cambiar el nombre del operador.
- Activar/desactivar las redes LTE, CDMA, GSM/WCDMA, etc.
- Modificar la configuración de IMS[15] (ej: VoLTE, VoWiFi).

Una lista más completa puede ser consultada en el sitio oficial de Android para desarrolladores, aunque la lista exhaustiva no es pública. Los privilegios de operador también abarcan los permisos sensibles de tiempo de ejecución, como los permisos para activar o desactivar servicios de datos, bloquear redes, seleccionar la red móvil actual de manera manual o automática, intercambiar comandos APDU con la SIMcard, etc. En el siguiente artículo profundizaremos en este último caso de uso.

Dentro de la app Android no se requiere ningún código adicional para hacer uso de los privilegios de operador. Estos privilegios son otorgados automáticamente si la SIMcard contiene el hash de la llave, con la cual se encuentra firmada la app, y opcionalmente el nombre del paquete de la app. Los privilegios también se revocan automáticamente si la SIMcard es removida.

Los privilegios de operador funcionan aun sin registro en la red móvil. También funcionan en modo avión. Si el teléfono soporta dual-SIM o multi-SIM, Android intenta escanear las firmas de ARA o ARF en todas las SIMcard disponibles.

Conclusiones

El manejo de permisos Android ha evolucionado y se ha vuelto cada vez más restrictivo. Aunque Android es un sistema operativo abierto, no todos los aspectos en el manejo de los permisos son públicos y están documentados.

Si tiene la necesidad de utilizar permisos sensibles y no puede obtener dichos permisos para la versión de Android más actual, es posible que en una versión más antigua la obtención de los mismos permisos sea menos restrictiva. Por ejemplo, la lectura del ID de la suscripción (IMSI) en Android 1.6 se podía hacer simplemente declarando el permiso `READ_PHONE_STATE` dentro del archivo *AndroidManifest.xml*.

Si está desarrollando una aplicación de uso masivo, es importante reducir el uso de permisos, especialmente los permisos sensibles. No hacerlo incrementa los riesgos de seguridad, por lo que también afecta la permanencia de la aplicación en las tiendas en línea como Google Play.

[15] https://es.wikipedia.org/wiki/Subsistema_Multimedia_IP



Tarjeta SIM como Módulo de Seguridad (HSM)

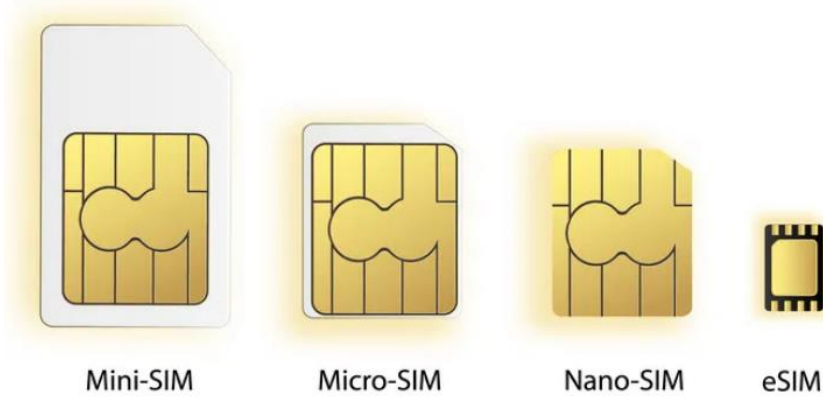
Victor Celer

El artículo anterior expuso que las *SIMcard* (o tarjetas SIM)[16] pueden utilizarse para otorgar privilegios de operador. Estos privilegios permiten obtener los permisos más elevados y sensibles en Android, sin recurrir al *rooteo*[17] del dispositivo. Por lo tanto, resulta evidente que tanto los operadores móviles como los proveedores de los dispositivos Android confían plenamente en la seguridad de la SIM.

Las SIM son dispositivos creados especialmente para resistir cualquier intento de *hackeo*. Estos dispositivos incluyen mecanismos de bloqueo automático para proteger la información sensible. Por ejemplo, los equipos de trazado y depuración para SIM pueden costar miles de euros, aunque estos equipos solamente interceptan los datos que viajan entre la SIM y el teléfono, como un «sniffer» de red.[18] Estos equipos no permiten clonar las tarjetas o extraer las claves allí almacenadas.

Módulos HSM

Dado que la SIM ofrece un alto nivel de seguridad, mayor que la protección alcanzable por medio de software, existen diferentes casos de uso de SIM en calidad de HSM (en inglés, *Hardware Security Module*).[19] Los módulos HSM son procesadores criptográficos, implementados en hardware, cuyo propósito es la protección de claves criptográficas. Aunque existen emuladores de HSM en software, los módulos HSM físicos pueden ofrecer mayor seguridad. Esto se debe a la imposibilidad de leer la memoria de estos procesadores de manera externa, lo cual hace prácticamente imposible la ingeniería inversa. En cambio, los módulos HSM emulados en software no pueden impedir la lectura de la memoria de manera física. El mismo razonamiento es usado por los fabricantes de SIM para argumentar que los chips físicos son más seguros que las SIM virtuales.



No se debe confundir las SIM virtuales con la evolución de eSIM.[20] Estas últimas virtualizan los datos contenidos en la SIM, pero el dispositivo de almacenamiento sigue siendo un chip físico, el cual

[16] https://es.wikipedia.org/wiki/Tarjeta_SIM
[17] https://es.wikipedia.org/wiki/Android_rooting
[18] https://es.wikipedia.org/wiki/Analizador_de_paquetes
[19] https://es.wikipedia.org/wiki/HSM
[20] https://es.wikipedia.org/wiki/ESIM

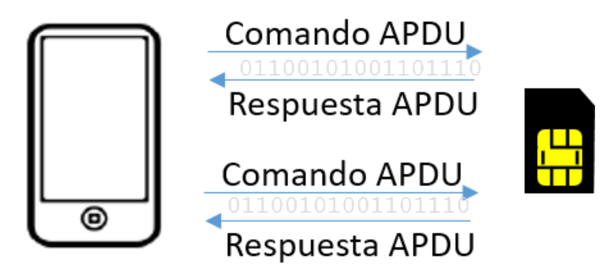
protege el acceso a la memoria de la misma manera como la SIM tradicional. Este chip no es removible. Por eso, se dice que eSIM es la versión embebida de SIM.

Las llaves de autenticación de la suscripción móvil se almacenan dentro de la SIM. Sin embargo, la SIM puede almacenar llaves adicionales. Por ejemplo, algunos bancos utilizan esta funcionalidad para cifrar las transacciones electrónicas hechas desde el celular. Las llaves bancarias son diferentes de las llaves de autenticación en la red móvil, pero se almacenan y protegen de la misma manera.

Comandos APDU

La comunicación entre el teléfono y la SIM funciona por medio de comandos APDU,[21] cuyo formato está definido en el estándar **ISO/IEC 7816-4**.[F]

Tradicionalmente la comunicación es iniciada por el teléfono y se realiza de manera síncrona. Es decir, se espera que la SIM responda antes de enviar el próximo APDU:



El comando APDU enviado hacia la SIM incluye un encabezado obligatorio con 4 octetos: CLA, INS, P1 y P2.

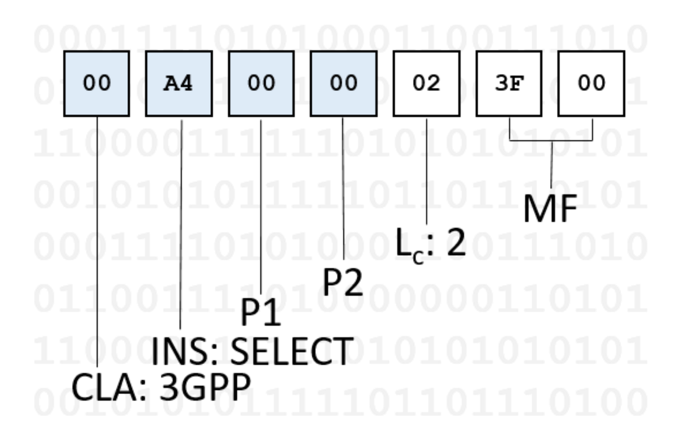
CLA	Clase: ETSI, 3GPP, Global Platform[G]
INS	Código de instrucción
P1	Parámetro #1
P2	Parámetro #2

El resto del APDU es opcional y puede incluir datos adicionales. Por ejemplo, estos datos adicionales sirven para seleccionar un archivo o un applet dentro de la SIM, enviar datos específicos para dicho applet, etc. El formato de los datos puede ser definido por el desarrollador, pero la longitud del APDU debe ser acorde al estándar ISO/IEC 7816-4.

Al procesar el APDU se produce una respuesta, cuyo formato establece que los 2 últimos octetos, conocidos como SW (en inglés, *Status Word*), indican el estado del resultado. Generalmente el valor `9000` en hexadecimal corresponde a un resultado exitoso. Veamos un ejemplo de APDU y su respectiva respuesta:

```
>> 00 A4 00 00 02 3F 00
<< 90 00
```

El contenido del APDU puede ser interpretado según el estándar **ETSI TS 102.221**,[H] como sigue:



El octeto CLA indica que la instrucción es de clase 3G. El octeto INS indica que se debe ejecutar la instrucción `SELECT` (seleccionar). Los parámetros 1 y 2 pueden indicar el modo de selección. En este caso se debe seleccionar el directorio de raíz conocido como MF (en inglés, *Master File*), cuyo identificador es `3F00`. El octeto L_C, el cual precede el identificador MF, indica la longitud de los datos adicionales. En este caso son 2 octetos adicionales, ya que el identificador MF ocupa 2 octetos. El directorio de raíz está presente en todos los

21 https://es.wikipedia.org/wiki/Application_Protocol_Data_Unit_(electr%C3%B3nica)

sistemas de archivos en la SIM. Por lo tanto, este APDU debería funcionar correctamente en cualquier SIM compatible con 3G.

Applets JavaCard

Supongamos que tenemos un applet criptográfico instalado en la SIM, el cual se usa para encriptar datos sensibles. Los applets se identifican por medio de un ID único conocido como AID. Supongamos que el AID de nuestro applet criptográfico de pruebas es `01020304050607`. Normalmente el AID tiene una longitud mayor, pero por simplicidad supondremos que son solamente estos 7 octetos. Entonces, el APDU para seleccionar dicho applet sería:

```
>> 00 A4 04 00 07 01 02 03 04 05 06 07 00
<< 90 00
```

Luego de seleccionar el applet podemos enviar otro APDU con los datos que deseamos encriptar.

Para mayor información acerca de los applets JavaCard y ejemplos de código se recomienda consultar el sitio oficial de Oracle JavaCard.[22] Los applets más simples pueden ser compilados con las herramientas gratuitas, disponibles en el mismo sitio, e instalados en una SIM de pruebas. La gran mayoría de las SIMc actuales soportan esta tecnología. Nótese que JavaCard es una tecnología basada en Java. El desarrollo de applets JavaCard tiene cierta similitud con el desarrollo de aplicaciones Java, excepto que el manejo de memoria es diferente debido a la escasez de este recurso en las SIM. El formato del bytecode JavaCard también se diferencia del formato de bytecode de JVM, pero estas diferencias son transparentes para el desarrollador.

Los applets JavaCard pueden hacer uso de la tecnología *SIM Application Toolkit*,[23] la cual permite ejecutar comandos proactivos en el teléfono, como enviar SMS, desplegar notificaciones y menú interactivos, entre otras funcionalidades.

Envío de APDU por medio de AT+CSIM

La manera más simple de poner en práctica el material expuesto hasta ahora consiste en utilizar un modem GSM para enviar los APDU por medio de un emulador de terminal, como *Hyper Terminal* o *PuTTY*. Con el mismo propósito se puede utilizar la librería gratuita *CelerCOM*[24] para Java.



De esta manera no es necesario utilizar un dispositivo lector de tarjetas y un software especializado para el envío de APDU. La mayoría de los modem GSM soportan el comando AT llamado `AT+CSIM`, el cual permite encapsular comandos APDU para la SIM. Internamente Android utiliza este mismo comando para ejecutar los APDU que las aplicaciones envían por medio del servicio `TelephonyManager`.[25]

Por ejemplo, el APDU para seleccionar el directorio raíz MF puede ser enviado de la siguiente manera:

```
AT+CSIM=14,"00a40000023f00"
```

El valor 14 es la longitud del APDU en caracteres.

---

[22] https://www.oracle.com/java/technologies/java-card/javacard-applet.html
[23] https://en.wikipedia.org/wiki/SIM_Application_Toolkit
[24] https://www.celersms.com/CelerCOM-es.htm
[25] https://developer.android.com/reference/android/telephony/TelephonyManager.html

Si el modem responde con un error genérico como «`+CME ERROR`» es posible que el modem no soporte el comando `AT+CSIM`. La mayoría de los modem GSM soportan este comando. Inclusive es posible utilizar un teléfono Android como modem, pero es más práctico usar un modem GSM para puerto USB.

Estos comandos `AT+CSIM`, enviados por medio de una terminal hacia el modem GSM, sirven para verificar la validez de los comandos. Se recomienda hacer estas pruebas antes de utilizar los mismos APDU en una aplicación Android.

Open Mobile API

Si ya instalamos el applet JavaCard en la SIM de pruebas y ya validamos los comandos APDU, el siguiente paso es hacer una aplicación Android para interactuar con la SIM.

Android soporta el envío de APDU por medio del servicio `TelephonyManager`[26] a partir de la versión 5.1. A partir de Android 9 esta funcionalidad está disponible por medio de `SEService`.[27] Esta nueva opción se basa en las especificaciones de **Open Mobile API v3**[I] de *Trusted Connectivity Alliance*, una organización integrada por los fabricantes de SIM, anteriormente conocida como *SIMAlliance*. Todas estas API requieren permisos elevados. Sin embargo, si estamos usando una SIM de pruebas, podemos instalar el hash de nuestra llave de desarrollo dentro de ARA o ARF para obtener privilegios de operador, como se explicó en el artículo anterior.

Existen implementaciones alternas, basadas en Open Mobile API. Una de las más conocidas se llama **SEEK** (en inglés, *Secure Element Evaluation Kit*).[J] Esta API fue creada por *Giesecke & Devrient*, uno de los fabricantes de SIMcard.

A continuación veamos un ejemplo que produce el mismo APDU que habíamos probado anteriormente con `AT+CSIM`, utilizando `TelephonyManager`:

```
import android.telephony.TelephonyManager;
import android.telephony.IccOpenLogicalChannelResponse;

   TelephonyManager tm =
      (TelephonyManager)getSystemService("phone");
   IccOpenLogicalChannelResponse resp =
      tm.iccOpenLogicalChannel(
         "01020304050607", // AID
         0                 // p2
   );
   int ch = resp.getChannel();
   if(ch > 0){
      String sResp =
         tm.iccTransmitApduLogicalChannel(
            ch,
            CLA, INS, P1, P2, P3, DATA
      );
      tm.iccCloseLogicalChannel(ch);
   }
```

Este código intenta seleccionar el applet JavaCard con AID `01020304050607`. Luego le envía el APDU correspondiente a los parámetros CLA, INS, P1, P2, P3 y DATA. El parámetro P3 es opcional. Si no aplica, se puede especificar un valor negativo para que el APDU se envíe solamente con los 4 octetos de CLA, INS, P1 y P2.

A veces, la SIMcard responde con SW `61XX`. Es decir, el penúltimo octeto de la respuesta tiene el valor hexadecimal 61. Esto significa que el APDU no puede ser procesado por el momento, pero se puede reintentar. Una implementación robusta debería procesar estas respuestas `61XX` y generar al menos un reintento de envío del mismo APDU. Es importante limitar los reintentos para no saturar la SIM y evitar un bloqueo indefinido. Un único reintento puede ser suficiente.

No olvide implementar manejo de errores. El anterior código debería quedar encerrado en un

26 https://developer.android.com/reference/android/telephony/TelephonyManager

27 https://developer.android.com/reference/android/se/omapi/SEService

bloque try/catch. Además, es importante recordar que estas API de TelephonyManager requieren el permiso MODIFY_PHONE_STATE[28]. Este permiso puede ser obtenidos en tiempo de instalación solamente hasta Android 2.2. A partir de Android 2.3 este permiso se puede obtener únicamente por las aplicaciones del sistema (preinstaladas) o por medio de privilegios de operador. Esta última opción es la más recomendada, como se había explicado más arriba.

A continuación se presenta otro ejemplo, utilizando Open Mobile API:

```
import android.se.omapi.SEService;
import android.se.omapi.Session;
import android.se.omapi.Channel;
import android.se.omapi.Reader;
import java.util.concurrent.Executors;

   ExecutorService exe =
      Executors.newSingleThreadExecutor();
   SEService se = new SEService(
      this, // context
      exe,  // para procesar los callback
      this  // listener
   );
   Reader[] readers = se.getReaders();
   if(readers.length > 0){
      Session sess = readers[0].openSession();
      Channel ch = sess.openLogicalChannel(
         // AID
         new byte[]{ 1, 2, 3, 4, 5, 6, 7 },
         0 // p2
      );
      byte[] respApdu = ch.transmit(
         new byte[]{ /* APDU */ }
      );
      ch.close();
   }
```

Nótese que en este caso los APDU son codificados como arreglos de bytes, no cadenas de texto hexadecimales. Otra diferencia importante es que la API puede manejar los reintentos de envío de APDU automáticamente, por lo que no es necesario procesar las respuestas con SW 61XX.

En este caso también se recomienda agregar un bloque try/catch para manejo de errores. Uno de los errores posibles es la ausencia de permisos.

Adicionalmente, se puede combinar los ejemplos de TelefonyManager y SEService para implementar una solución compatible con diferentes versiones de Android.

Conclusiones

La SIM, como módulo de seguridad, puede ser útil para almacenar datos sensibles, implementar servicios de encripción y desencripción, entre otros usos. La ventaja principal tiene que ver con la seguridad física de la SIM.

Las aplicaciones Android requieren permisos elevados para poder interactuar con la SIM. Para obtener estos permisos se recomienda hacer uso de los privilegios de operador, los cuales pueden ser otorgados por medio de la misma SIM.

Existen diferentes API para interactuar con la SIM en Android. La disponibilidad de estas API depende del fabricante y la versión del sistema operativo. Es posible combinar las API para soportar múltiples versiones de Android.

[28] https://developer.android.com/reference/android/Manifest.permission#MODIFY_PHONE_STATE



Sistemas Abiertos, pero Seguros

Vladimir Kameñar

Android se posicionó como el sistema operativo móvil más utilizado en el mundo, en gran parte debido a su calidad de sistema abierto. Los sistemas operativos más antiguos, como Symbian, Blackberry OS, Windows Mobile, con los cuales entró a competir Android a mediados de los años 2000, eran sistemas de código cerrado. Google logró cautivar a los desarrolladores ofreciendo una plataforma abierta, personalizable, con herramientas de desarrollo gratuitas.

Existe la opinión de que los sistemas abiertos son menos seguros. Si los hacker tienen acceso a los códigos fuente de los componentes del sistema operativo, entonces no tienen la necesidad de hacer ingeniería inversa en busca de vulnerabilidades. Si las aplicaciones tienen control sobre la mayoría de los recursos del dispositivo, entonces es posible que estos recursos se utilicen en perjuicio de los intereses del usuario. Uno de los grandes retos para Google ha sido mantener Android como una plataforma abierta, pero al mismo tiempo contrarrestar sus posibles desventajas en materia de seguridad.

Tim Cook dijo en *VivaTech 2021* que ha habido 47 veces más incidencias de malware en Android que en iOS.[29] Panda Security había publicado una métrica similar en 2019, indicando que los casos de malware en Android son cerca de 50 veces más frecuentes que en iOS.[30]

En realidad, estas cifras no son tan desfavorables para Android, como podría parecer a primera vista. Estas métricas abarcan teléfonos *rooteados* y versiones de Android muy antiguas, en las cuales los niveles de seguridad eran mínimos. Además, si Android es tan popular entre los desarrolladores, también lo es entre los hacker y los autores de malware.

Algunos de los sistemas operativos más seguros corresponden a la familia BSD: OpenBSD, NetBSD, FreeBSD.[K] Estos sistemas operativos son abiertos, pero aun así son más seguros que muchos de los sistemas operativos cerrados.

Una de las ventajas del código fuente abierto, desde el punto de vista de seguridad, es que los expertos en seguridad pueden estudiar este código. Esto no solo ayuda a identificar y corregir posibles vulnerabilidades. También permite detectar patrones de diseño que conducen a la ocurrencia de vulnerabilidades futuras.[L]

[29] *A live conversation with Tim Cook and Brut*, VivaTech 2021 https://www.youtube.com/watch?v=RMHclZR6Neo

[30] *Android devices 50 times more infected with malware compared to iOS*, Panda Security (2019) https://www.pandasecurity.com/en/mediacenter/mobile-security/android-more-infected-than-ios/

Permisividad vs. inseguridad

Las primeras versiones de Android tenían controles de seguridad mínimos. Las aplicaciones podían acceder a todos los datos del usuario y el teléfono. Desde el punto de vista del desarrollador esto era una gran ventaja en comparación con otros sistemas operativos, pero al mismo tiempo provocó que los fabricantes de teléfonos (Samsung, HTC, Sony, etc.) comenzaran a personalizar sus versiones de Android, eliminando las funcionalidades más peligrosas o restringiendo el acceso a las mismas.

Google también comenzó a eliminar las funcionalidades que representaban mayor riesgo. Por ejemplo, uno de los cambios polémicos fue la eliminación de la API `sendRawPdu` en Android 2.x,[31] sin explicación oficial de parte de Google y sin alternativa que no implique *rooteo* del dispositivo. Esta API era usada por las aplicaciones para enviar SMS binarios.

La primera generación de Android permitió evidenciar un conflicto de intereses. Por una parte, Google deseaba que Android se convirtiera en la plataforma móvil más utilizado, desplazando a la competencia. Por otra parte, Google no podía permitir que los desarrolladores tuvieran control total sobre el dispositivo, ya que esto último hacía inviable el uso comercial masivo de Android.

En este punto se vuelve evidente que la vulnerabilidad original de Android se debió a su permisividad (falta de restricciones para acceder a los recursos, datos o funcionalidades sensibles), no al hecho de que el sistema fuese abierto. Esta permisividad comenzó a desaparecer en las versiones subsiguientes. Las versiones más actuales tienen niveles de permisividad comparables con los de otros sistemas operativos, no solamente móviles. Por ejemplo, Android 8 introdujo restricciones para las aplicaciones que se ejecutan en segundo plano.[32] Estas restricciones ayudan a ahorrar el uso de batería, datos y otros recursos, por parte de los procesos que no son visibles para el usuario. Otro ejemplo de reducción de permisividad son las restricciones para el acceso a la SIM a partir de Android 2.x. Actualmente solamente las aplicaciones del sistema y las aplicaciones con privilegios de operador pueden interactuar directamente con la SIM.

Cada nueva versión de Android introduce restricciones adicionales. Esto obliga a los desarrolladores a modificar su código para adaptarse a las restricciones nuevas. En algunos casos no es posible encontrar una alternativa equivalente, como en el ejemplo de `sendRawPdu` mencionado más arriba. Por lo tanto, cualquier aplicación que hace uso de recursos sensibles puede volverse obsoleta en una futura versión de Android.

Diferentes sabores de Android

Por ejemplo, un teléfono de marca Samsung y otro teléfono de marca Huawei pueden tener la misma versión de Android, pero las plataformas no son idénticas. Como se mencionó más arriba, los fabricantes de teléfonos comenzaron a personalizar sus propias distribuciones, conocidas como *Custom ROM*,[33] prácticamente a partir de las primeras versiones de Android.



[31] https://issuetracker.google.com/issues/36917186

[32] *Límites de ejecución en segundo plano* https://developer.android.com/about/versions/oreo/background

[33] *List of custom Android distributions* (en inglés) https://en.wikipedia.org/wiki/Custom_ROM

Las versiones genéricas, publicadas y mantenidas por Google, son conocidas como AOSP (en inglés, *Android Open Source Project*).[34] Estas versiones son usadas en los dispositivos producidos por Google y otros dispositivos genéricos, generalmente de bajo costo.

Las versiones personalizadas pueden incluir cambios cosméticos, ya que las primeras versiones genéricas no contaban con interfaces gráficas particularmente atractivas. Por ejemplo, los dispositivos de gama alta fabricados por HTC incluían unas mejoras de interfaz de usuario conocidas como *Sense*. Las versiones personalizadas también pueden incluir cambios en el *kernel* para soportar mejor el hardware del dispositivo. Esto permite mejorar el rendimiento y aprovechar mejor los recursos del dispositivo. El hardware es bastante heterogéneo, por lo que la versión AOSP no siempre logra tener compatibilidad y aprovechar el hardware de la mejor forma posible. Por ejemplo, algunos fabricantes incluyen termovisor, lector RFID, impresora, entre otros ejemplos de dispositivos que no son de uso común en los teléfonos móviles. Para soportar estas funcionalidades añadidas también puede ser necesario personalizar Android.

Las versiones personalizadas pueden ser más seguras o menos seguras. Por ejemplo, algunos operadores preinstalan aplicaciones, las cuales permiten agregar aplicaciones adicionales sin el consentimiento del usuario. Esto compromete la privacidad y seguridad del usuario. Además, estos mecanismos de instalación automática de apps pueden ser explotados para difundir malware, burlando todas las restricciones de seguridad del dispositivo.

En resumen, no todos los Android son iguales. Por lo tanto, resulta sorprendente que diferentes fuentes generalicen el concepto de seguridad en Android sin hacen un estudio comparativo de diferentes versiones y personalizaciones existentes.

Un reto para el desarrollador

El desarrollador Android puede elegir que su aplicación solamente funcione en dispositivos no *rooteados* con las últimas versiones de Android disponibles. Algunos desarrolladores de servicios bancarios y afines toman esta decisión para reducir los riesgos de vulnerabilidades. Ingresar al banco desde un teléfono muy viejo puede ser como acceder desde un computador público. El riesgo de comprometer la seguridad por cuenta de un programa espía puede ser muy alto.

Por otra parte, soportar una mayor variedad de dispositivos, incluyendo los más antiguos, permite alcanzar una mayor base de usuarios.

En caso de que el usuario llegue a ser víctima de fraude, la responsabilidad no va a ser del fabricante del teléfono, el sistema operativo o el operador móvil. El desarrollador de la app será señalado como responsable. Por lo tanto, es muy importante evaluar los riesgos antes de publicar la aplicación.

[34] https://source.android.com/

Referencias

[A] Bruce Schneier (2011). «*Secrets and Lies: Digital Security in a Networked World*», p.257, John Wiley & Sons

[B] Adnan Ali, Vasaki Ponnusamy, Anbuselvan Sangodiah (2019). «*User Behaviour-Based Mobile Authentication System*», DOI:10.1007/978-981-13-6861-5_40

[C] Hassan Sbeyti (2016). «*Mobile user authentication based on user context and behavioral pattern (MOUBE)*», Arab Open University.

[D] «*Secure Element Access Control v1.1*», Global Platform (2014)

[E] «*Cryptographic Token Information Format Standard v1.1*», RSA Laboratories (1999)

[F] «*ISO/IEC 7816-4*», ISO/IEC JTC 1/SC 17 (2005)

[G] «*Global Platfrom Technology Card Specification v2.3.1*», Global Platform (2018)

[H] «*Smart Cards; UICC-Terminal interface; Physical and logical characteristics (Release 15)*», ETSI (2018)

[I] «*Open Mobile API v3.0*», Trusted Connectivity Alliance (2016).

[J] Michael Roland, Michael Hölzl (2016). «*Open Mobile API: Accessing the UICC on Android Devices*» (en inglés), University of Applied Sciences Upper Austria.

[K] Michael W. Lucas (2013), «*Absolute OpenBSD: Unix for the practical paranoid*» (2ª ed), No Starch Press. ISBN 978-1-59327-476-4

[L] Amiangshu Bosu, Jeffrey C. Carver et al (2014). «*When are OSS Developers More Likely to Introduce Vulnerable Code Changes? A Case Study.*» (en inglés), IFIP International Conference on Open Source Systems, DOI:10.1007/978-3-642-55128-4_37

"La puerta mejor cerrada es aquella que puede dejarse abierta"

Proverbio chino

